

Understanding Multicore Cache Behavior of Loop-based Parallel Programs via Reuse Distance Analysis

Meng-Ju Wu and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park
{mjwu,yeung}@umd.edu

Abstract

Understanding multicore memory behavior is crucial, but can be challenging due to the cache hierarchies employed in modern CPUs. In today's hierarchies, performance is determined by complex thread interactions, such as interference in shared caches and replication and communication in private caches. Researchers normally perform simulation to sort out these interactions, but this can be costly and not very insightful. An alternative is reuse distance (RD) analysis. RD analysis for multicore processors is becoming feasible because recent research has developed new notions of reuse distance that can analyze thread interactions. In particular, concurrent reuse distance (CRD) models shared cache interference, while private-stack reuse distance (PRD) models private cache replication and communication.

Previous multicore RD research has centered around developing techniques and verifying accuracy. In this paper, we apply multicore RD analysis to better understand memory behavior. We focus on loop-based parallel programs, an important class of programs for which RD analysis provides high accuracy. First, we develop techniques to isolate thread interactions, permitting analysis of their relative contributions. Then, we use our techniques to extract several new insights that can help architects optimize multicore cache hierarchies. One of our findings is that data sharing

in parallel loops varies with reuse distance, becoming significant only at larger RD values. This implies capacity sharing in shared caches and replication/communication in private caches occur only beyond some capacity. We define C_{share} to be the “turn-on capacity” for data sharing, and study its impact on private vs. shared cache performance. In addition, we find machine scaling degrades locality at smaller RD values and increases sharing frequency (i.e., reduces C_{share}). We characterize how these effects vary with core count, and study their impact on the preference for private vs. shared caches.

1 Introduction

Memory performance is a major determiner of overall system performance and power consumption in multicore processors. For this reason, understanding how applications utilize the on-chip cache hierarchies of multicore processors is extremely important for architects, programmers, and compiler designers alike. But gaining deep insights into multicore memory behavior can be very difficult due to the complex cache organizations employed in modern multicore CPUs.

Today, a typical multicore processor integrates multiple levels of cache on-chip with varying capacity, block size, and/or set associativity across levels. In addition, cores may physically share cache. Each core normally comes with its own dedicated L1 cache, but the caches further down the hierarchy can be either private or shared. Moreover, data replication is usually permitted across the private caches, with hardware cache coherence used to track and manage replicas.

In such cache hierarchies, memory performance depends not only on how well per-thread working sets fit in the caches, but also on how threads’ memory references interact. Many complex thread interactions can occur. For instance, inter-thread memory reference interleaving leads to *interference* in shared caches. Also, *data sharing* may reduce aggregate working set sizes in shared caches, partially offsetting the performance-degrading interference effects. But in private caches, data sharing causes *replication*, increasing capacity pressure, as well as *communication*.

To study these complex effects, researchers normally rely on architectural simulation [1, 2, 3, 4, 5, 6, 7, 8], but this can be very costly given the large number of simulations required to explore on-chip cache design spaces. A more efficient approach that can potentially yield deeper insights is *reuse distance (RD) analysis*. RD analysis measures a program’s memory reuse distance histogram—*i.e.*, its locality profile—directly quantifying the application-level locality responsible for cache performance. Locality profiles report reference frequency *across the continuum of RD values*, which can enable powerful analyses. For example, a single locality profile can predict the cache-miss count at every possible cache capacity without having to simulate all of the configurations.

Multicore RD analysis is relatively new, but is becoming a viable tool due to recent work. In particular, researchers have developed new notions of reuse distance that account for thread interactions. *Concurrent reuse distance (CRD)* [9, 10, 11, 12, 13, 14] measures RD across thread-interleaved memory reference streams, and accounts for interference and data sharing between threads accessing shared caches. *Private-stack reuse distance (PRD)* [11, 12] measures RD separately for individual threads using per-thread coherent stacks. PRD accounts for replication and communication between threads accessing private caches. Together, CRD and PRD profiles (*i.e.*, locality histograms based on CRD/PRD) can analyze shared and private cache performance.

One problem with multicore RD is that it is sensitive to memory interleaving, and hence, *architecture dependent*. A CRD/PRD profile measured on one architecture may not be valid for other architectures if memory interleaving changes. But such profile instability is minimal when threads exhibit similar access patterns [10, 11, 14]. For example, programs exploiting loop-level parallelism contain *symmetric threads* that tend to execute at the same rate regardless of architectural assumptions. For such programs, CRD/PRD profiles are essentially architecture independent and provide accurate analyses. For the general case, however, CRD/PRD profiles can exhibit error.

Despite its shortcomings, RD analysis can provide extremely rich information about multicore

memory behavior. But surprisingly, there has been very little work on extracting insights from this information, as current research has focused primarily on developing techniques and verifying their accuracy. In this paper, we use RD analysis to better understand multicore memory behavior. In particular, we extract new insights that can potentially help architects optimize multicore cache hierarchies. Our work also demonstrates the kind of analyses that are possible.

For accuracy, we focus on loop-based parallel programs with architecture independent CRD and PRD profiles. Though somewhat restrictive, our study is still fairly general given the pervasiveness of parallel loops. They dominate all data parallel codes. They also account for most programs written in OpenMP, one of the main parallel programming environments. And while our results come from parallel loops, we believe many of our insights apply to parallel programs in general.

Our work makes several contributions. First, we review different thread interactions in both shared and private caches, and show how basic CRD and PRD profiles capture them. Then, we isolate the different effects to analyze their relative contributions. We leverage existing techniques for shared caches [14], but develop new techniques for private caches. We apply this analysis to loop-based parallel programs, and show how different thread interactions contribute to overall CRD/PRD behavior. One of our findings is that data sharing varies with reuse distance. For parallel loops, we find sharing typically does not occur at small RD, but can increase at larger RD.

Second, we explore the architectural implications of our data sharing insights. In particular, we define “ C_{share} ” to be the RD value (*i.e.*, cache capacity) at which data sharing becomes noticeable. We then study the impact of C_{share} on the relative performance of private vs. shared caches. We find shared caches potentially make sense beyond C_{share} where increased sharing leads to lower miss counts compared to private caches. But this benefit must be weighed against the higher access latency of shared caches, whose impact depends on the miss-rate of the upstream cache. We evaluate this tradeoff using our CRD/PRD profiles for a simple tiled processor architecture.

Third, we use RD analysis to study the impact of machine scaling on memory behavior, showing how CRD and PRD profiles evolve at different core counts. We corroborate Wu’s findings [14] that scaling core count for loop-based parallel programs degrades shared cache locality, but only at smaller capacities. Using Wu’s C_{core} metric [14], we quantify such “scaling containment” across several benchmarks. In addition, we show scaling-induced locality degradation is more comprehensive for private caches, and we also demonstrate C_{share} shifts to smaller capacities with scaling. Finally, we study the impact of these machine scaling insights on private vs. shared cache performance.

The rest of this paper is organized as follows. Section 2 reviews thread interactions, and presents techniques to isolate their effects. Then, Section 3 uses our techniques to analyze loop-based parallel programs, and applies insights to analyze private vs. shared cache performance. Next, Section 4 develops machine scaling insights. Finally, Sections 5 and 6 discuss related work and conclusions.

2 Multicore RD Analysis

Reuse distance is the number of unique memory references performed between two references to the same data block. An RD profile—*i.e.*, the histogram of RD values—can analyze uniprocessor cache performance. Because a cache of capacity C can satisfy references with $RD < C$ (assuming LRU), the number of cache misses is the sum of all reference counts in an RD profile above the RD value for capacity C . In multicore CPUs, RD analysis must consider memory references from simultaneously executing threads. *Per-thread RD profiles* can be measured using independent LRU stacks, but other techniques are needed to account for thread interactions. Below, we review different thread interactions (Section 2.1), and then develop techniques to isolate their effects (Section 2.2).

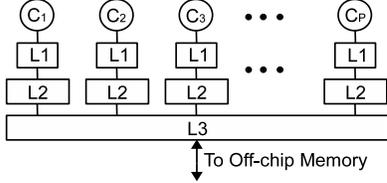


Figure 1. Multicore cache hierarchy.

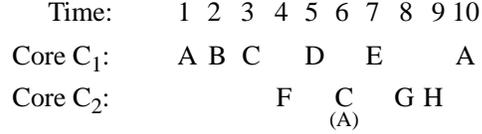


Figure 2. Two interleaved memory reference streams, illustrating different thread interactions.

2.1 CRD and PRD Profiles

Multicore processors often contain both shared and private caches. For example, Figure 1 illustrates a typical CPU consisting of 2 levels of private cache backed by a shared last-level cache. Threads interact very differently in each type of cache, requiring separate locality profiles.

Concurrent reuse distance (CRD) profiles report locality across thread-interleaved memory reference streams, thus capturing interference in shared caches. CRD profiles can be measured by assuming an interleaving between threads’ memory references, and applying the interleaved stream on a single (global) LRU stack [10, 11, 12, 14]. For example, Figure 2 illustrates a 2-thread interleaving in which core C_1 touches blocks A – E , and then re-references A , while core C_2 touches blocks C and F – H . Figure 3a shows the state of the global LRU stack when C_2 re-references A . Notice, C_2 interferes with C_1 ’s reuse of A : per-thread $RD = 4$ becomes $CRD = 7$. $CRD > RD$ because some of C_2 ’s references (F – H) are distinct from C_1 ’s references, causing *dilation* in CRD.

Private-stack reuse distance (PRD) profiles report locality across per-thread memory reference streams that access coherent private caches. PRD profiles can be measured by applying threads’ references on private LRU stacks that are kept coherent. Without writes, the private stacks do not interact, so $PRD = \text{per-thread } RD$. For example, Figure 3b shows the PRD stacks corresponding to Figure 3a assuming all references are reads. For C_1 ’s reuse of A , $PRD = RD = 4$. Note, however, the multiple private stacks still contribute to increased cache capacity. (Here, the capacity needed to satisfy $PRD = 4$ is 10, assuming 2 caches with 5 cache blocks each). To capture this effect, we compute the *scaled PRD* (sPRD) which equals $T \times PRD$, where T is the number of threads.

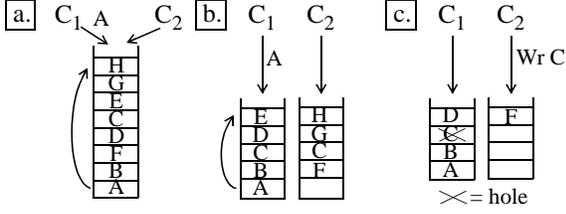


Figure 3. LRU stacks showing (a) dilation and overlap in CRD profiles, (b) scaling, replication, and (c) holes in PRD profiles.

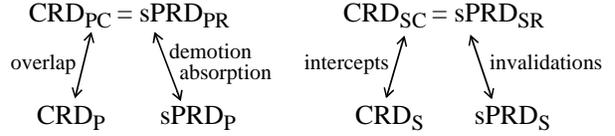


Figure 4. Quantifying individual thread interaction effects.

In addition to dilation and scaling, multicore RD analysis also takes data sharing into account. In CRD profiles, data sharing offsets dilation by introducing *overlap*. For example, in Figure 2, while C_2 's reference to C interleaves with C_1 's reuse of A , this does not increase A 's CRD because C_1 already references C in the reuse interval. On top of overlap, data sharing also affects CRD via *intercepts*. For example, if C_2 references A instead of C at time 6 in Figure 2, then C_1 's reuse of A has $CRD = 3$, so CRD actually becomes *less* than RD.

In PRD profiles, read sharing causes *replication*, increasing overall capacity pressure. Figure 3b illustrates this, showing duplication of C in the private stacks. Because scaling aggregates private LRU stack contents, replication is automatically captured in sPRD profiles. In contrast, write sharing causes *invalidation*. For example, if C_2 's reference to C is a write instead of a read, then invalidation would occur in C_1 's stack,¹ as shown in Figure 3c. To prevent invalidations from promoting blocks further down the LRU stack, invalidated blocks become *holes* rather than being removed from the stack [11]. Holes are unaffected by references to blocks above the hole (*e.g.*, if C_1 were to reference D in Figure 3c), but a reference to a block below the hole moves the hole to where the referenced block was found (*e.g.*, if C_1 were to reference B , D would be pushed down and the hole would move to depth 3, preserving A 's stack depth).

Of course, invalidations reduce locality for victimized data blocks since reuse of these blocks will always cache miss (*i.e.*, coherence misses). But invalidations can also improve locality because

¹Existing PRD techniques perform invalidation to maintain inter-stack coherence [11, 12].

the holes they leave behind eventually absorb stack demotions. For example, in Figure 3c, C_1 's subsequent reference to E does not increase A 's stack depth because D will move into the hole. (Since this is E 's first reference, in effect, the hole moves to $\text{RD} = \infty$). Hence, C_1 's reuse of A has PRD (sPRD) = 3 (6) instead of 4 (8). We call this effect *demotion absorption*.

2.2 Quantifying Thread Interactions

To provide insights into multicore memory behavior, we separately quantify dilation, overlap, and intercepts in shared caches and scaling, demotion absorption, and invalidations in private caches by creating several new locality profiles that isolate these thread interactions. In particular, we borrow CRD profile techniques from [14], and extend them for PRD profiles. Later, in Section 3, we acquire all of the isolation profiles, and use them to study multicore cache behavior.

Figure 4 illustrates our approach. First, we acquire profiles for references to private versus shared data separately. (Note, we still keep all data in the same LRU stacks from Figure 3; we only register computed CRD/PRD values in separate profiles). We call the private profiles CRD_P and sPRD_P , and the shared profiles CRD_S and sPRD_S . CRD_P and sPRD_P do not include interactions directly associated with shared data—*i.e.*, intercepts and invalidations, respectively. So, CRD_P shows dilation and overlap effects while sPRD_P shows scaling and demotion absorption effects. CRD_S and sPRD_S primarily show intercept and invalidation effects, respectively. They also contain the other effects to some degree, but as we will see, this is not a problem in most cases.

Next, we isolate the sharing-based interactions. For CRD profiles, we remove sharing by prepending every memory reference's address with the ID of the core performing the reference. That way, references to the same data block by different cores appear as distinct blocks in the global LRU stack. We call the profiles acquired on such CID-extended stacks CRD_{PC} and CRD_{SC} . CID extension removes overlap in private profiles, so comparing CRD_P and CRD_{PC} separately quantifies

overlap. Similarly, CID extension removes intercepts in shared profiles, so comparing CRD_S and CRD_{SC} separately quantifies intercepts. This is shown in Figure 4.

For sPRD profiles, we remove write sharing to isolate the hole-related interactions. This is done by converting all writes into reads. We call the profiles for such read-only stacks $sPRD_{PR}$ and $sPRD_{SR}$. Read conversion removes demotion absorption in private profiles, so comparing $sPRD_P$ and $sPRD_{PR}$ separately quantifies demotion absorption. Similarly, read conversion removes invalidations in shared profiles, so comparing $sPRD_S$ and $sPRD_{SR}$ separately quantifies invalidation (see Figure 4). Finally, after CID extension and read conversion, CRD and sPRD profiles only exhibit dilation and scaling, respectively. By comparing these profiles to per-thread RD profiles, we can separately quantify dilation and scaling (not shown in Figure 4).

3 Thread Interactions Analysis

This section applies the isolation techniques introduced in Section 2.2 to study thread interactions in loop-based parallel programs. We first discuss how we acquire profiles, then present our interaction insights, and finally, apply our insights to analyze private vs. shared cache performance.

3.1 Profile Acquisition

We use Intel PIN [15] to acquire locality profiles. Our PIN tool maintains independent LRU stacks to compute per-thread RD profiles, a global LRU stack to compute a CRD profile, and coherent private LRU stacks to compute an sPRD profile. All stacks employ 64-byte memory blocks, the cache block size we assume. Our PIN tool follows McCurdy and Fischer’s method [16], performing functional execution with context switching between threads every memory reference, which interleaves threads’ references uniformly in time. While this does not account for timing, it is accurate for loop-based parallel programs. In particular, Wu [14] shows CRD profiles acquired this way can predict shared cache MPKI for loop-based parallel programs to within 10% of simulation.

		FFT	LU	Radix	Barnes	FMM	Ocean	Water	KMeans	BlackS.
Problem Sizes	S1	2^{18}	512^2	2^{19}	2^{15}	2^{15}	258^2	16^3	2^{18}	2^{18}
	S2	2^{22}	2048^2	2^{23}	2^{19}	2^{19}	1026^2	40^3	2^{22}	2^{22}
Insts (M)	S1	129	344	93	1,015	1,006	107	143	742	242
	S2	2,420	22,007	1,687	19,145	16,570	1,636	2,099	11,874	3,867

Table 1. Parallel benchmarks used in our study.

Others have shown similar accuracy for both CRD and PRD profiles on parallel loops [10, 11].

Because the thread interactions we study occur within individual parallel loops, we acquire profiles on a per-loop basis. In our benchmarks, parallel loops often begin and end at barriers, so we record profiles between every pair of barrier calls—*i.e.*, per parallel region. Multiple loops can occur in each parallel region so this doesn’t separate all loops, but it is sufficient for our study.²

Within parallel regions, we acquire the isolation profiles from Section 2.2. To create private vs. shared profiles, we record each memory block’s CRD (PRD) values separately as well as the number of times the block is referenced by each core. After a parallel region completes, we assess sharing: if a single core performs 90% or more of a block’s references, the block is private; otherwise, it is shared. We then accumulate all memory blocks’ CRD (PRD) counts into either CRD_P ($sPRD_P$) or CRD_S ($sPRD_S$) based on their observed sharing. (We also similarly create private and shared versions of per-thread RD profiles, RD_P and RD_S). To create no-sharing profiles, we maintain a second global LRU stack for which we apply CID extension and a second set of coherent private LRU stacks for which we apply read conversion. Using the same method to separate private and shared references, we compute CRD_{PC} , CRD_{SC} , $sPRD_{PR}$, and $sPRD_{SR}$ from these stacks.

Our study employs 9 benchmarks, each running 2 problem sizes. Table 1 lists the benchmarks: FFT, LU, RADIX, Barnes, FMM, Ocean, and Water from SPLASH2 [17], KMeans from MineBench [18], and BlackScholes from PARSEC [19]. Table 1 specifies the problem sizes, S1 and S2, in terms of data elements. We acquire profiles in each benchmark’s parallel region. Table 1 also reports parallel region instruction count when profiling a sequential execution of the benchmark.

²To facilitate parallel region profiling, we instrument PIN to recognize each program’s barrier function.

3.2 Analysis

We analyze a specific example, the Barnes benchmark. While different interactions occur to varying degrees across benchmarks, we find all parallel regions exhibit the same behaviors, so our insights for Barnes are generally applicable. (We will consider all benchmarks in Section 3.3). Figure 5 shows different CRD and PRD profiles for the most important parallel region in Barnes running on 16 cores. Each profile plots reference count (Y-axis) versus RD (X-axis). RD values are multiplied by the block size, 64 bytes, so each X-axis reports RD as capacity. For each profile, reference counts from multiple adjacent RD values are summed into a single RD bin, and plotted as a single Y value. For capacities 0–128KB, bin size grows logarithmically; beyond 128KB, all bins are 128KB each.

Private Profiles Analysis. Figure 5a compares CRD_{PC} and $sPRD_{PR}$ against RD_P . As described in Section 2.2, CRD_{PC} and $sPRD_{PR}$ do not contain sharing-induced interactions, so this comparison shows dilation and scaling in isolation. In Figure 5a, we see $sPRD_{PR}$ is roughly a 16x scaling of RD_P . This is somewhat expected since $sPRD$ is simply a scaled version of private-stack profiles. More surprisingly, CRD_{PC} is essentially identical to $sPRD_{PR}$, and is also a 16x scaling of RD_P . This is due to the regular memory interleaving exhibited by parallel loops. In a shared cache, intra-thread reuses at a particular distance RD tend to experience dilation by a factor RD from each of the other simultaneous threads because all of the threads are symmetric. Consequently, dilation degrades locality at the same rate as scaling: linear with the number of threads.

While dilation and scaling are equivalent, the sharing-related interactions in shared vs. private caches are not. To illustrate, Figure 5b plots CRD_P and $sPRD_P$. As described in Section 2.2, these profiles quantify overlap and demotion absorption, respectively, when compared against CRD_{PC} (and equivalently, $sPRD_{PR}$). From Figures 5a-b, we see several important behaviors.

First, CRD_P and $sPRD_P$ both terminate well before CRD_{PC} : 6.4MB for CRD_P and 15MB for

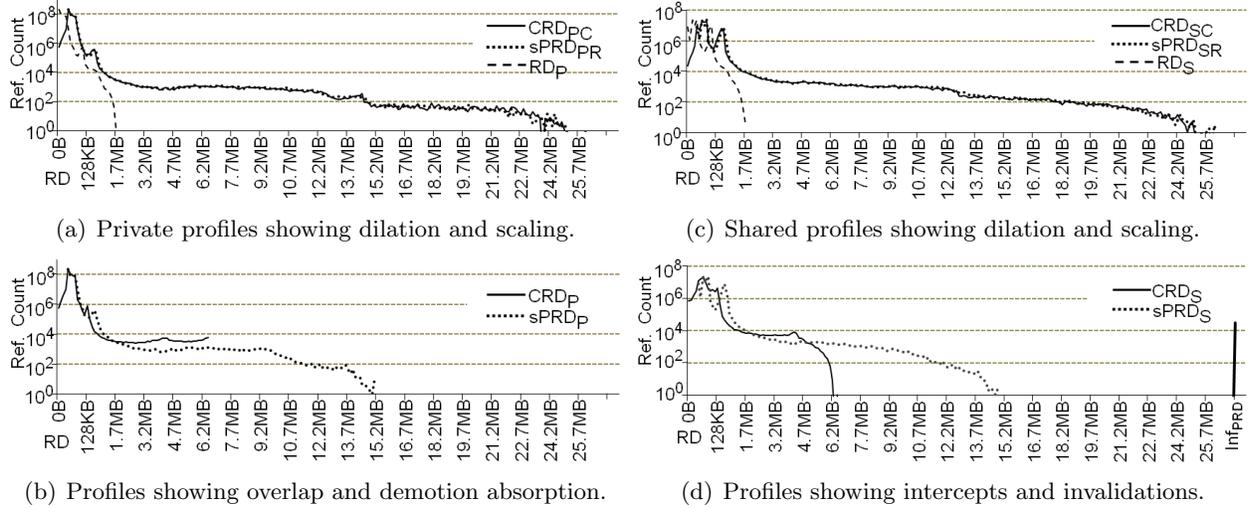


Figure 5. Barnes' locality profiles for the most important parallel region on 16 cores.

sPRD_P compared to 24.9MB for CRD_{PC}. As discussed in Section 2.1, data sharing introduces overlap that offsets dilation in CRD profiles; sharing also causes demotion absorption that reduces scaling in PRD profiles. These effects contract CRD_P and sPRD_P relative to CRD_{PC}. Second, the fact that CRD_P and sPRD_P terminate at different capacities is significant. It shows dilation is more dominant than demotion absorption since the former causes greater contraction. (We find write sharing is relatively infrequent, so demotion absorption tends to be a minor effect). But more importantly, the discrepancy also quantifies replication. As mentioned in Section 2.1, sPRD profiles aggregate replicated data whereas CRD profiles do not. Hence, the difference in termination between CRD_P and sPRD_P—about 9.9MB in Figure 5b—measures the volume of replicated data.

Lastly, CRD_P and PRD_P's contraction varies with reuse distance. Because contraction is caused by sharing, its presence or absence along CRD_P and sPRD_P permits *assessing the degree of data sharing as a function of reuse distance*. For example, in Figures 5a-b, we see CRD_P, sPRD_P, and CRD_{PC} are coincident at small RD values. Then, at around 2KB, contraction begins and the profiles diverge. Contraction grows with increasing RD, and finally causes the different termination mentioned above. In other words, data sharing is absent at capacities below 2KB, but increases from 2KB onwards. This makes sense since programmers tend to partition parallel loops to reduce

sharing frequency. Later, we will see the significance of such distance-based sharing variation.

Shared Profiles Analysis. Figures 5c-d plot the shared profiles, which exhibit similar behavior to the corresponding private profiles in Figures 5a-b. In particular, Figure 5c shows CRD_{SC} and $sPRD_{SR}$ are essentially identical, and are both a 16x scaling of RD_S —*i.e.*, dilation and scaling are equivalent for parallel loops in the absence of sharing. Moreover, Figure 5d shows CRD_S and $sPRD_S$ contract compared to $CRD_{SC}/sPRD_{SR}$, with CRD_S contracting the most. Similar to Figure 5b, this illustrates overlap, demotion absorption, and the volume of replicated data.

Figures 5c-d also show intercept and invalidation effects. As discussed in Section 2.1, intercepts disrupt intra-thread reuse by splitting reuse windows. For example, Figure 2 shows an intercept bisecting C_1 's reuse, making $CRD = 3$. But if the intercept occurs at time 9, $CRD = 0$, and if the intercept occurs at time 2, $CRD = 6$. Because the interception point is random, intercepts tend to “spread” CRD profiles, causing distortion. This effect is visible in Figure 5d, particularly at small RD values. In contrast, invalidations move references to $RD = \infty$. The rightmost point on the X-axis in Figure 5d, labeled “ Inf_{PRD} ,” reports the number of invalidations in $sPRD_S$. Unlike intercepts, invalidations do not cause distortion. This is partly due to the fact that invalidations leave holes which preserve stack depth for non-invalidated blocks.

Notice, the reference counts in Figures 5c-d are much smaller than Figures 5a-b, about 6x less. In our benchmarks, we find private profiles dominate shared profiles. Hence, dilation and overlap in CRD profiles along with scaling and demotion absorption in PRD profiles determine overall shared/private cache performance.

3.3 Private vs. Shared Caches

The memory behaviors from Section 3.2 suggest opportunities to optimize multicore caches. We now study one example, selecting between private vs. shared caches, a classic multicore design deci-

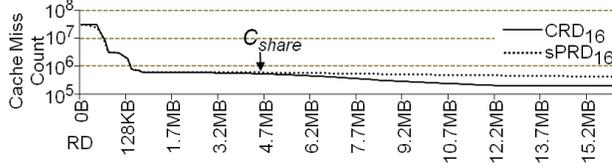


Figure 6. CRD.CMC and sPRD.CMC profiles for FFT running the S1 problem on 16 cores.

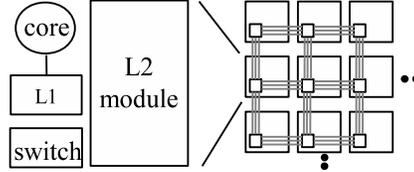


Figure 7. Tiled multicore processor.

sion [20, 21]. (We present basic insights in this section, and then extend them later in Section 4.2). For parallel programs with data sharing, shared caches more efficiently utilize cache capacity by keeping only a single copy of threads’ overlapping working sets in cache. However, private caches more effectively mitigate access latency by caching each thread’s working set physically close to the accessing core. Multicore RD analysis can evaluate this tradeoff precisely.

The extent to which shared caches more efficiently utilize capacity depends on the degree of data sharing in an application. As shown in Section 3.2, this varies with reuse distance. *Hence, a shared cache’s capacity utilization advantage over a private cache depends on its capacity.* Figure 6 illustrates this point by plotting cache-miss count (CMC) versus RD for both the CRD and sPRD profiles from FFT (running the S1 problem on 16 cores). These profiles, which we call CRD.CMC and sPRD.CMC, respectively, are computed as $\text{CRD.CMC}[i] = \sum_{j=i}^{N-1} \text{CRD}[j]$ and $\text{sPRD.CMC}[i] = \sum_{j=i}^{N-1} \text{sPRD}[j]$, where N is the total number of bins in each profile.

Figure 6 shows private and shared caches incur the same number of cache misses at small capacities. As discussed in Section 3.2, there is no data sharing in this region, so dilation and scaling are equivalent and CRD and sPRD are coincident. Beyond a certain capacity, the shared organization reduces cache misses compared to the private organization. As discussed in Section 3.2, this is the data sharing region where overlap contracts CRD more than demotion absorption contracts PRD. This region is also where replication accumulated via scaling of sPRD profiles (and to a lesser extent invalidations) causes private caches to have increased cache misses compared to shared caches.

We define “ C_{share} ” to be the capacity where an application begins to exhibit data sharing such

	FFT	LU	Radix	Barnes	FMM	Ocean	Water	KMeans	BlackScholes
S1	4.4M	32K	3.6M	2K	4K	768K	2K	8K	1K
S2	76.8M	64K	72M	2K	27.8M	91.7M	2M	16K	1K

Table 2. C_{share} for the S1 and S2 problem sizes running on 16 cores.

that CRD shows an advantage over sPRD (we require CRD to be 10% lower than sPRD). Figure 6 illustrates C_{share} . In Table 2, we report C_{share} across all of our benchmarks for the S1 and S2 problems running on 16 cores. As Table 2 shows, C_{share} is highly application dependent, varying between 1KB and 4.4MB for the S1 problem. For the S2 problem, C_{share} can exceed 70MB.

To study the impact of capacity utilization on private vs. shared cache performance, we consider an architecture that can be easily configured with private or shared caches: the tiled processor in Figure 7. In a typical tiled processor, each tile contains a core, a private L1 cache, and an L2 module. The L2 module can either be a second private cache, or a slice of a shared cache formed by all L2 modules across the tiled processor. A point-to-point 2D mesh network connects the tiles, carrying either private L2 cache requests or shared L2 slice requests. Due to the 2D topology, we assume network messages incur $\sqrt{T} + 1$ hops on average, where T is the number of tiles. The average memory access time (AMAT) for the tiled CPUs with private and shared L2 caches are:

$$AMAT_p = L1_{lat} + L2_{lat_p} \times \frac{sPRD_CMC[T \times L1_{size}]}{sPRD_CMC[0]} + DRAM_{lat} \times \frac{sPRD_CMC[L2_{size}]}{sPRD_CMC[0]} \quad (1)$$

$$AMAT_s = L1_{lat} + L2_{lat_s} \times \frac{sPRD_CMC[T \times L1_{size}]}{sPRD_CMC[0]} + DRAM_{lat} \times \frac{CRD_CMC[L2_{size}]}{sPRD_CMC[0]} \quad (2)$$

A shared cache is best when $AMAT_p > AMAT_s$. Given the above equations, this occurs when:

$$(sPRD_CMC[L2_{size}] - CRD_CMC[L2_{size}]) \times DRAM_{lat} > sPRD_CMC[T \times L1_{size}] \times (L2_{lat_s} - L2_{lat_p}) \quad (3)$$

Equation 3 shows shared caches are better when the memory stall savings they provide via the $sPRD_CMC/CRD_CMC$ gap in Figure 6 (LHS of Equation 3) exceeds private caches' savings in L2 access latency (RHS of Equation 3). Notice, the access latency savings is weighted by the L2 access

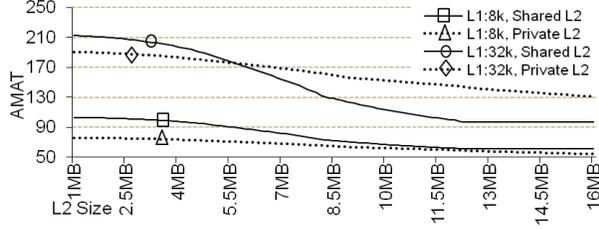


Figure 8. FFT's L2 $AMAT_p$ and L2 $AMAT_s$ for different L1 and L2 capacities.

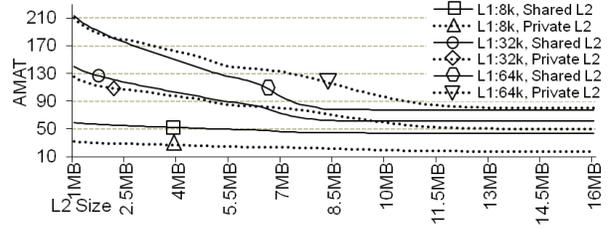


Figure 9. FMM's L2 $AMAT_p$ and L2 $AMAT_s$ for different L1 and L2 capacities.

frequency—*i.e.*, the L1 caches' miss count. So, preference for private or shared not only depends on the capacity of the cache being optimized, but also on the capacity of the *upstream cache*.

To illustrate, Figures 8 and 9 plot L2 $AMAT_p$ and $AMAT_s$ as a function of total L2 size for the FFT and FMM benchmarks (both run the S1 problem on 16 cores). Different pairs of curves show results for different L1 sizes. When computing L2 AMAT, we assume a 3-cycle per-hop network latency, a 10-cycle L2 module latency, and a 200-cycle DRAM latency. These graphs show several insights. First, at small L1 capacities, the RHS of Equation 3 dominates, so private caches are always better. This occurs in Figures 8 and 9 when the L1 is 8KB. Second, for larger L1 caches, the RHS of Equation 3 reduces, allowing the LHS to dominate if the $sPRD_CMC/CRD_CMC$ gap is sufficiently large. As discussed earlier, a gap opens up only beyond C_{share} . Hence, under larger L1 caches, private is still best below C_{share} , but above C_{share} , shared can outperform private. This occurs in Figure 8 for the 32KB L1 with a private-to-shared L2 cross-over at 5.8MB, and in Figure 9 for the 32KB and 64KB L1s with L2 cross-overs at 6.4MB and 2.2MB, respectively.

Finally, at large L2 capacities, private and shared cache performance tends to converge because there are much fewer total L2 misses. This diminishes shared cache advantage. In fact, private caches may regain a performance advantage if the off-chip stalls become insignificant compared to the L2 access volume. For example, Figure 9 shows a shared-to-private L2 cross-over at 9.7MB.

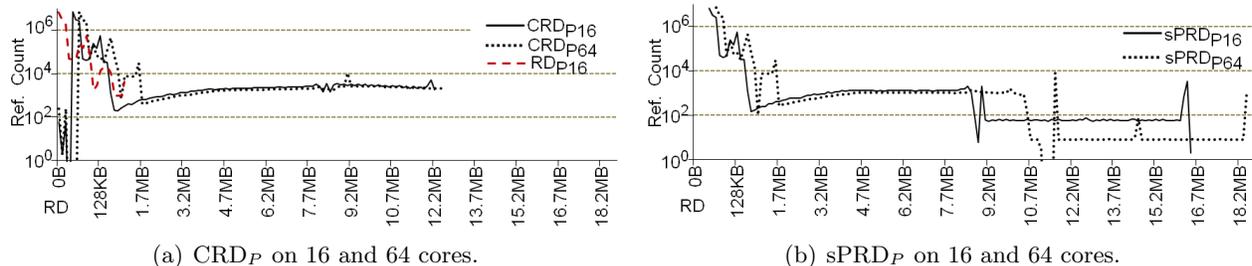


Figure 10. FFT’s locality profiles for the most important parallel region on 16 and 64 cores.

4 Machine Scaling Analysis

This section applies our isolation techniques introduced in Section 2.2 to study machine scaling’s impact on loop-based parallel programs. We first present insights, and then discuss their architectural implications. Similar to Section 3, we present the detailed analysis on a single benchmark, this time FFT, and then generalize to other benchmarks.

4.1 Scaling CRD and sPRD Profiles

Figure 10a compares the 16- and 64-core CRD_P for the FFT benchmark running the S1 problem. RD_P is also shown. As shown in Section 3.2, CRD_P exhibits two effects: dilation at small RD due to memory interleaving of private data references, and overlap that offsets dilation at larger RD due to data sharing. The dilation effect increases with core count. At small RD, dilation scales CRD linearly compared to RD. So, increasing the number of cores causes a proportional increase in dilation. This effect is visible in Figure 10a: the 16- and 64-core CRD_P at small RD are 16x and 64x scalings of RD_P. Hence, the 4x machine scaling shifts CRD_P in the small-RD region by 4x.

Although dilation increases with machine scaling, overlap still offsets its effects at larger RD. As shown in Section 3.2, this contracts CRD_P relative to CRD_{PC} (not shown in Figure 10a), causing sub-linear shift in the large-RD region. Interestingly, in Figure 10a we see that not only does overlap occur, but its contraction of the 16- and 64-core CRD_P makes both profiles eventually track each other and terminate at the same RD value. Co-termination is a reflection of the fact that machine

scaling usually does not increase the total data in the application much, so maximum RD is roughly constant with core count. This analysis shows machine scaling degrades locality for shared caches, but its impact is limited to smaller capacities. It has almost no impact at large capacities.

Figure 10b compares the 16- and 64-core $sPRD_P$ for the same FFT benchmark. Because $sPRD_P$ is just a scaled version of private stack profiles, $sPRD_P$ shifts to larger RD as core count increases. But as Figure 10b shows, while the 64-core $sPRD_P$ is a 4x shift of the 16-core $sPRD_P$ at small RD, it is only a 1.1x shift at large RD. This is due to the fact that individual threads from parallel loops execute fewer loop iterations as core count increases, which tends to reduce long-distance intra-thread reuse. This in turn truncates per-thread PRD, and reduces $sPRD_P$ scaling.³ So, like CRD_P , $sPRD_P$ also shifts non-uniformly. But our analysis shows machine scaling degrades locality in private caches more than in shared caches since $sPRD_P$ is affected across all cache capacities.

In addition to CRD_P and $sPRD_P$, machine scaling also impacts shared profiles. In particular, more cores lead to higher intercept rates, causing greater spreading of CRD_S . More cores also lead to higher invalidation rates, moving more references to $RD = \infty$. But as shown in Section 3.2, private profiles dominate shared profiles in the parallel regions we’ve studied. Hence, the effects shown in Figure 10 largely determine overall CRD and $sPRD$ behavior under machine scaling.

4.2 Architectural Implications

Machine scaling’s impact on locality profiles has several implications for multicore cache hierarchy design. In this section, we discuss locality degradation mitigation in shared caches and data sharing under scaling. Then, we revisit private vs. shared cache selection to include machine scaling effects.

As described in Section 4.1, the locality degradation for shared caches is limited to smaller cache sizes. This implies shared caches beyond a certain capacity will see no impact as core count increases, in essence “containing” the negative effects of machine scaling. In previous work, Wu [14]

³Demotion absorption also contracts $sPRD_P$, but as discussed in Section 3.2, this effect is comparatively minor.

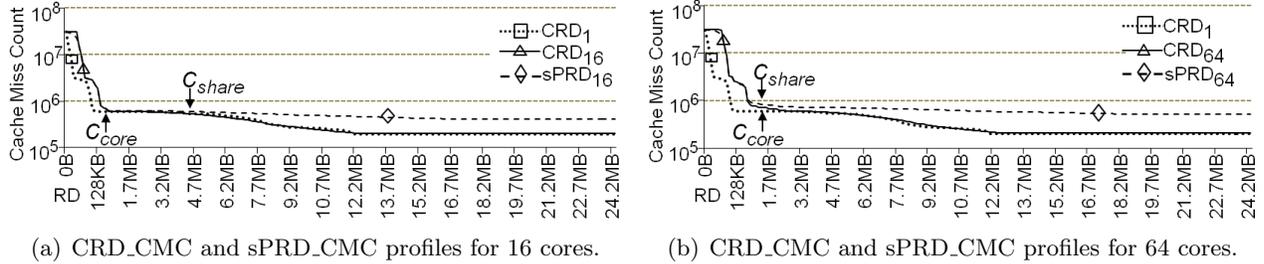


Figure 11. C_{core} and C_{share} relationship at 16 and 64 cores for FFT running the S1 problem.

has also observed this phenomenon, and proposed the parameter, “ C_{core} ,” to quantify the point of scaling containment. Figure 11a illustrates C_{core} for FFT. Wu computes C_{core} by comparing CRD_CMC at some core count (e.g., 16 cores in Figure 11a) against the CMC profile for RD (the 1-core locality profile). At small RD values, CRD_CMC incurs noticeably more cache misses than RD, but eventually, the two CMC profiles merge due to the overlap effects discussed in Section 4.1. C_{core} is defined to be this merge point.⁴ Figure 11a shows $C_{core} = 210\text{KB}$ for scaling up to 16 cores.

To illustrate detailed behavior, Figure 12 shows C_{core} across all of our benchmarks. Each graph in Figure 12 reports C_{core} when scaling between 2–256 cores for a particular benchmark. The two solid curves in each graph show C_{core} variation on the S1 and S2 problems separately. One result from Figure 12 is that C_{core} increases with machine scaling. This is due to the continued shifting of CRD_P (and hence, CRD) at larger core counts, as discussed in Section 4.1. Nevertheless, Figure 12 shows C_{core} is relatively small. For the S1 problem, C_{core} never exceeds 2.2MB while for the S2 problem, C_{core} is within 19.6MB. Although continued problem scaling will undoubtedly increase C_{core} , we find that in many cases, machine scaling can be contained by modest shared caches.

In contrast to shared caches, Section 4.1 shows machine scaling’s locality impact in private caches occurs across all cache sizes. This implies the cache miss gap between private and shared caches increases with machine scaling. To illustrate, Figures 11a and b plot sPRD_CMC on top of CRD_CMC for 16 and 64 cores, respectively. Figure 11 shows scaling from 16 to 64 cores

⁴Wu uses a minimum threshold gap to pinpoint C_{core} similar to what we use for C_{share} .

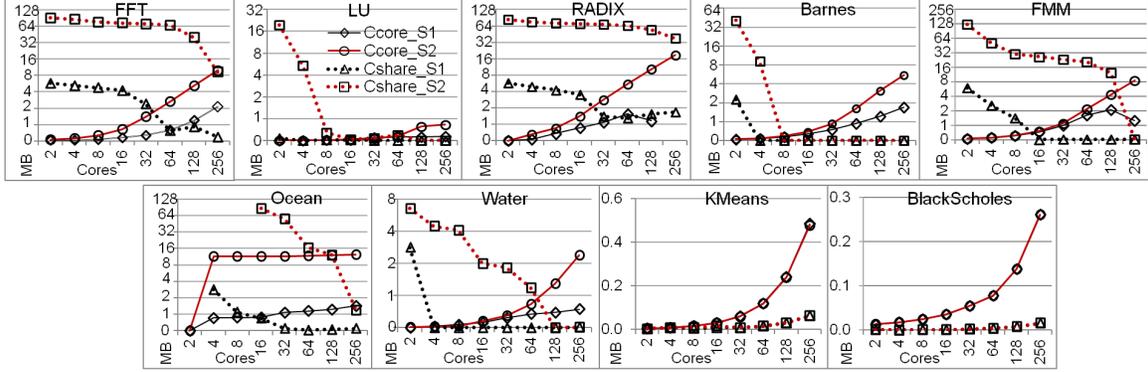


Figure 12. C_{core} and C_{share} relationship at 2-256 cores running the S1 and S2 problems.

indeed increases sPRD_CMC relative to CRD_CMC, especially beyond C_{core} where CRD_CMC is insensitive to machine scaling.

Not only does machine scaling increase the cache miss gap, it also moves the sharing point, C_{share} , to smaller RD. Comparing Figures 11a and b, we see that as sPRD_CMC increases relative to CRD_CMC, their divergence point shifts to the left. This makes sense: distributing the same amount of work across more cores is likely to increase sharing frequency. Figure 12 confirms this for our benchmarks. Alongside C_{core} , Figure 12 reports C_{share} for 2–256 cores on the S1 and S2 problems. For 7 of 9 programs, Figure 12 shows machine scaling always reduces C_{share} . (For the other 2 programs, Kmeans and Blackscholes, C_{share} remains about the same with machine scaling).

4.2.1 Scaling Private-vs-Shared Caches

We now revisit the private vs. shared cache selection problem to extend the architectural insights from Section 3.3 by incorporating the machine scaling effects discussed above. We again consider L2 $AMAT_p$ and L2 $AMAT_s$ for our example tiled CPU. To account for machine scaling, we use the sPRD_CMC and CRD_CMC profiles at different core counts discussed in Section 4.1.

Figure 13 presents the results. In Figure 13, we plot two graphs per benchmark: the top graph shows tiled CPUs with 16KB L1s while the bottom graph shows tiled CPUs with 64KB L1s. Within each graph, L2 capacity is varied from 0–128MB along the X-axis, and machine scaling is studied

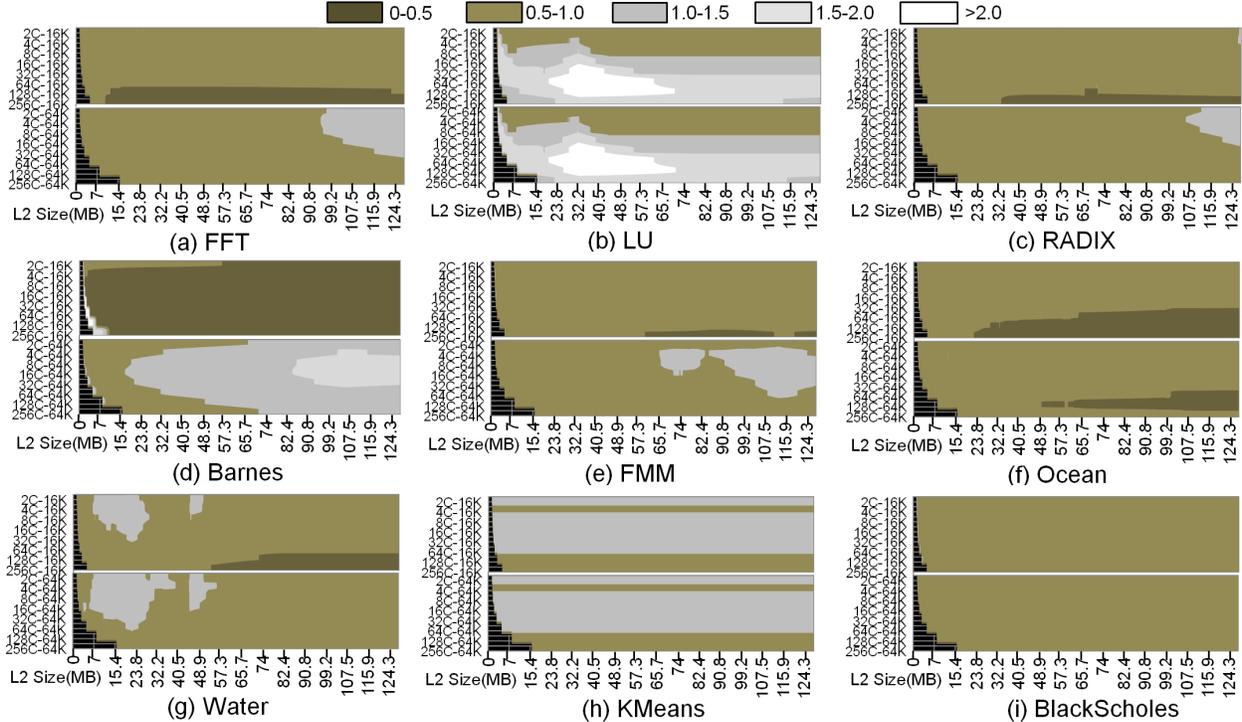


Figure 13. Private vs. shared performance across L1 capacity, L2 capacity, and machine scaling.

along the Y-axis for 2–256 cores. For each CPU configuration, the ratio $\frac{L2\ AMAT_p}{L2\ AMAT_s}$ is plotted as a gray scale. The legend reports the gray scale assignments. In particular, the 3 lightest gray scales (ratio > 1.0) indicate shared caches are best. We do not consider CPUs with less total L2 cache than total L1 cache; these cases are shaded black in Figure 13. All benchmarks run the S2 problem.

All basic insights from Figures 8 and 9 are also visible in Figure 13. In particular, L2 shared caches are best only when conditions make the LHS of Equation 3 dominant. First, the L1 cache capacity must be sufficiently large to reduce L2 access frequency. For example, Figure 13 shows a 16KB L1 is not large enough for FFT, RADIX, and FMM, so private caches are always best in these cases. Second, the L2 capacity must be larger than C_{share} so that the sPRD_CMC/CRD_CMC gap is non-zero. Indeed, we see all configurations for which shared caches are best in Figure 13 do occur beyond their corresponding C_{share} value in Figure 12. Notice for FFT, RADIX, FMM, and OCEAN, private caches usually prevail. For these benchmarks, C_{share} is large (see Figure 12), so shared cache benefit does not occur at reasonable capacities. And third, not only should the

sPRD_CMC/CRD_CMC gap be non-zero, it must also be large enough to provide significant memory stall savings. For example, in Barnes, C_{share} is small and the sPRD_CMC/CRD_CMC gap is uniformly large, so shared caches dominate across most L2 capacities (as long as L1 capacity is 64KB). But in Water, while C_{share} is also small, the sPRD_CMC/CRD_CMC gap varies across L2 capacity, so preference changes from private to shared and back to private again. LU also shows interesting sPRD_CMC/CRD_CMC gap variation, though shared caches are usually best.

Figure 13 also shows machine scaling’s impact on private vs. shared cache selection. As shown in Figure 11, machine scaling shifts both CRD_CMC and sPRD_CMC linearly with core count at small RD, which can place greater capacity pressure on L1 caches. As we add cores to our tiled CPU, we also increase total L1 capacity linearly, canceling this effect. But machine scaling also increases the physical extent of the CPU, making shared L2 accesses more costly than private L2 accesses. Hence, under linear L1 capacity scaling, the shared L2 access penalty (RHS of Equation 3) actually increases with core count. These scaling trends tend to make private caches more desirable at larger core counts. Figure 13 shows this effect in FFT with 16KB L1s, in RADIX, and in OCEAN.

Counterbalancing this effect, Figure 11 also shows machine scaling shifts C_{share} to smaller capacities and increases the sPRD_CMC/CRD_CMC gap. These scaling trends tend to make shared L2s more desirable at larger core counts. In Figure 13, we see this effect in LU and Barnes.

5 Related Work

Our research is closely related to Wu’s work on predicting CRD profiles across machine scaling [14]. In particular, Wu was the first to isolate dilation, overlap, and intercept effects in CRD profiles. Our CRD_P , CRD_{PC} , and CRD_S profiles are borrowed directly from Wu. Also, Wu was the first to observe the limited impact of machine scaling on shared caches, proposing the C_{core} metric to quantify the affected capacities. In our research, we extend Wu’s thread interactions analysis for private caches. Specifically, we identify scaling, replication, and communication effects, and

propose the $sPRD_P$, $sPRD_{PR}$, $sPRD_S$ and $sPRD_{SR}$ profiles to isolate them. We are also the first to compare different interactions across CRD and PRD profiles to develop data sharing insights. In addition, all of the architecture implications we develop (other than C_{core}) are completely novel.

There has also been significant research on acquiring locality profiles. Ding and Chilimbi [9], Jiang *et al* [10], and Xiang *et al* [13] present techniques to construct CRD profiles from per-thread RD profiles. By composing per-thread statistics, these techniques avoid acquiring interleaved memory reference streams, simplifying the profiling process. However, analysis cost can be significant as the numerous ways in which threads' memory references can interleave must be considered.

Schuff *et al* [11] acquire locality profiles by simulating uniform memory interleaving of simultaneous threads (much like our approach), and evaluate cache-miss prediction accuracy using the acquired profiles. In addition to predicting shared cache performance using CRD profiles, Schuff also acquires PRD profiles, and predicts private cache performance. (Our PRD profiling methodology is modeled after Schuff's approach). In subsequent work, Schuff also speeds up profile acquisition via sampling and parallelization techniques [12].

We leverage these previous techniques to acquire our CRD and PRD profiles. However, the focus of our work is on the insights that can be drawn from analyzing the profiles, and on the insights' implications for multicore cache hierarchy design. In particular, to our knowledge, we are the first to define a reuse distance-based measure of data sharing, and to show its implications for on-chip cache organization.

6 Conclusions

This paper investigates memory behavior for loop-based parallel programs using multicore RD analysis. Our study isolates different thread interactions in shared and private caches, analyzing their relative contributions. For loop-based parallel programs, we find data sharing varies with reuse distance—in particular, it does not occur below a certain RD value. We define C_{share} to be the

“turn-on capacity” for sharing, and measure it across several benchmarks. Then, we show private vs. shared cache performance depends on both C_{share} and the miss frequency of the upstream cache. We evaluate this relationship using our CRD/PRD profiles for a simple tiled processor. Finally, we study the impact of machine scaling on CRD and PRD profiles as well as C_{share} . We find machine scaling both degrades locality at smaller RD values and increases sharing frequency. We then show how these scaling trends impact private vs. shared cache design. Our study not only reveals several new insights about multicore memory behavior, but we believe it also demonstrates multicore RD analysis is a powerful tool that can help computer architects improve future multicore designs.

References

- [1] J. Davis, J. Laudon, and K. Olukotun, “Maximizing CMP Throughput with Mediocre Cores,” in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.
- [2] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, “Exploring the Cache Design Space for Large Scale CMPs,” *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.
- [3] J. Huh, S. W. Keckler, and D. Burger, “Exploring the Design Space of Future CMPs,” in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.
- [4] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures,” in *Proceedings of the International Symposium on Microarchitecture*, 2009.
- [5] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron, “CMP Design Space Exploration Subject to Physical Constraints,” in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.
- [6] J. Li and J. F. Martinez, “Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2005.
- [7] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang, and Y. Solihin, “Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling,” in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.
- [8] L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell, “Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design,” in *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*, 2007.
- [9] C. Ding and T. Chilimbi, “A Composable Model for Analyzing Locality of Multi-threaded Programs,” Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

- [10] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, “Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?,” in *Proceeding of Compiler Construction*, 2010.
- [11] D. L. Schuff, B. S. Parsons, and J. S. Pai, “Multicore-Aware Reuse Distance Analysis,” Tech. Rep. TR-ECE-09-08, Purdue University, 2009.
- [12] D. L. Schuff, M. Kulkarni, and V. S. Pai, “Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.
- [13] X. Xiang, B. Bao, C. Ding, and Y. Gao, “Linear-time Modeling of Program Working Set in Shared Cache,” in *Proceedings of the 20th International Symposium on Parallel Architectures and Compilation Techniques*, (Galveston Island, TX), October 2011.
- [14] M.-J. Wu and D. Yeung, “Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs,” in *Proc. of the 20th International Symp. on Parallel Architectures and Compilation Techniques*, (Galveston Island, TX), October 2011.
- [15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.
- [16] C. McCurdy and C. Fischer, “Using pin as a memory reference generator for multiprocessor simulation,” *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.
- [17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.
- [18] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, “MineBench: A Benchmark Suite for Data Mining Workloads,” in *Proceedings of the International Symposium on Workload Characterization*, 2006.
- [19] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.
- [20] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, “A NUCA Substrate for Flexible CMP Cache Sharing,” in *Proceedings of the International Conference on Supercomputing*, (Boston, MA), June 2005.
- [21] B. A. Nayfeh and K. Olukotun, “Exploring the Design Space for a Shared-Cache Multiprocessor,” in *Proceedings of 21st International Symposium on Computer Architecture (ISCA-21)*, (Chicago, IL), pp. 166–175, ACM, February 1994.